CS 211 End of Year Project Write Up

The Problem: Find the shortest route to 1000 airports provided every airport visited is at least 100km away from the previous one.

The Language: While researching this project, I experimented with the idea of using C# and a Nvidia CUDA library to run it off my GPU. Given the time constraints, learning a new architecture was impractical. I ultimately chose Java over C# because I am more comfortable using Java. While the JVM will always be slower than running on native hardware, it became a choice of what I was more familiar and skilled with. So, Java won out.

The Strategy: When planning this project, I looked at the obvious solution, brute force. For this problem, brute force has a complexity of 1000 factorial. While this would no doubt yield the best answer, it is impractical. While thinking on the brute force I thought of the possibility of utilising some form of GPU acceleration, but this ultimately went nowhere. Following some research, I settled on the nearest neighbour heuristic as the basis for my algorithm. Early implementations all ran into the same problem. I would get to the second last destination, then have a massive jump to the start position. To work around this, I though about the idea of going on a continent by continent route. So, this dividing idea became the foundation of the algorithm I used. I wanted to create a unique solution that could be scaled.

The Algorithm: My algorithm takes a more unique approach, I tried experimenting with a divide and conquer strategy. So, once I take in my co-ordinates, I create a lookup table for the distances between cities. I then divide the earth up into 8 sectors based roughly off continent. I then determine roughly where I want to enter and exit the zone and based off that I select the point closest to that side (north/south/east/west). This division into 8 sectors aids in optimisation, while the algorithm is still O(n^3) by definition, in practice the algorithm is more similar to that of an O(n^2) because the 3rd n dependent loop is only called when a dead end needs to be avoided. And because of the 8 sectors, my n^2 loops are called 8 times for smaller fractions of n. This massively cuts down on runtime. In terms of the pathfinding, I use a nearest neighbour-based algorithm. But there's a twist, with my 8 sectors and their entry/exit points I find a nearest neighbour path from start-middle and end-middle. This double-barrelled nearest neighbour prevents a situation where you could have a very neat path with a large jump to the exit point. My nearest neighbour finds the next nearest points from the start and end paths and then adds them to the list. To prevent clashes and revisiting areas, I use a boolean array to keep track of visited areas. This nearest neighbour has a problem, due to the 100km constraint, it can run into dead ends where no valid city exits. Now, I could simply add more cities to my route, but that wastes time. Instead I have a dead-end flag built into my next city method. By default, the next city index is -1, if the search cannot find a valid city, it returns -1. I then call my dead-end fix method. This method takes the current city and finds the 2 closest cities in my current path it can be slotted between. I then use a basic array shift loop to open the gap and insert my current city into that gap. I then take my new current city (at index 'i' of the array) and try find a

valid city from there. This loop repeats until the next city is valid. To aid in optimisation, I also have it run this shift and slot method if the distance between the current city and next city is long, this is in an attempt to keep my path as short as possible. This pathfinding is carried out 8 times for the 8 sectors. Once each of the sectors has been mapped, I put them through a randomised optimisation loop that takes 2 random cities, compares the total distance if I swapped them, if this is a shorter path, it swaps them. I then have a non-randomised optimisation loop that works in the same way except it steps through the cities 1 by 1. Once the sub-path has been generated, it is added to the overall route. This route is rigged to start in western Europe and end in eastern Europe. I can then use the same optimisation loop on the overall path. This cleans up the loose edges. Once I have my full route, I can pass it into a quick method to convert my co-ordinates into a numerical route order based off the initial inputs since my algorithm does not use the original order. Total runtime is approximately 13 minutes. Runtime complexity is O(n^3) but in practice it would be closer to O(n^2). I can run it for longer to yield better results. Result accuracy varies due to random elements, but tends to hover around the 430,000km distance.

Data Structure: My data structure is simple, it's an n*n 2d array to handle distances. While not the fastest, it is simple, reliable, and easily scalable. I also use a 2d array to hold co-ordinates. I know that I have a lot of redundant space since most of the arrays could be reused, but this was mostly a biproduct of debugging.

Conclusion: My code runs, it produces valid outputs. Due to the semi random nature of the optimisation there exists a possibility that my algorithm will create the best path. I feel that doing this project I learned a lot about running into dead ends with ideas, but it was great to experiment and implement ideas.